Scala for Professionals

Mastering The Type System

Type Parameters

```
case class Tuple2[A, B]
trait Set[A] {
  def map[B](f: A => B): Set[B]
}
```

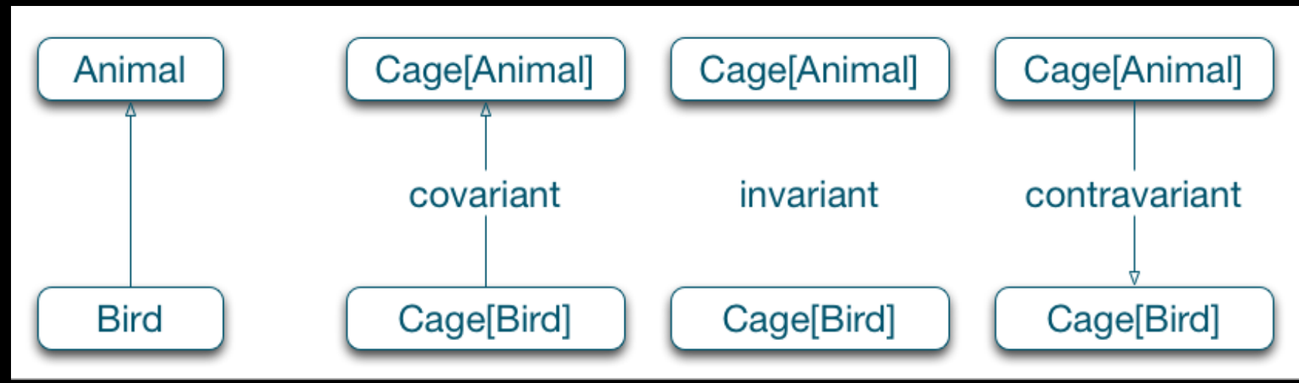
- Classes, traits, and methods can accept type parameters. This is known as **parametric polymorphism**[†].
- We accept one or more **type parameters**, declared and provided in square brackets – `[ & ]`.
- **Idiomatic Convention**: Type parameters should be declared as single capital letters, usually starting with A and proceeding in alphabetic order.

[†] A very fancy way of saying "the behavior changes based on the type of a parameter"

Type Variance

Type parameters introduce the question of variance:

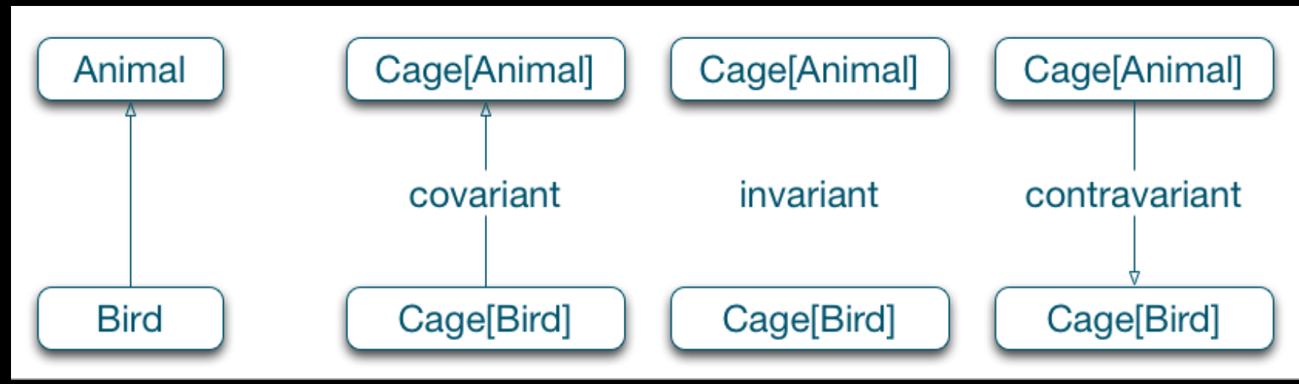
- ‘Given two types with a subtype relationship, what's the relation of a type parameter of each of those?’



Type Variance

‘Given two types with a subtype relationship, what's the relation of a type parameter of each of those?’

- None: Invariant
- Aligned: Covariant
- Opposed: Contravariant



Variance and Mutability

```
val cage: Cage[Animal] = new Cage[Bird] // OK if covariant
cage.put(new Elephant) // Oops!
```

- At first glance, variance rules result from mutability:
  - Read-only (immutable) => Covariant
  - Write-only => Contravariant
  - Read + Write (mutable) => Invariant
- **Example**: Try stuffing an Elephant into a Bird Cage.
- **Quiz**: Why wouldn't this example compile?

Invariant as the Default

```
val cage: Cage[Animal] = new Cage[Bird]
/*
<console>:10: error: type mismatch;
 found   : Cage[Bird]
 required: Cage[Animal]
 Bird <: Animal, but class Cage is invariant in type A.
 You may wish to define A as +A instead. (SLS 4.5)
*/
```

- By default, parameterized types are invariant.
- Take note that Scala provides us with a descriptive error message.

Variance Declarations

By using **variance declarations**, we can change the relationship of subtypes (variance) *per* type parameter.

- We use the + modifier to declare a type parameter as **covariant**:

```
class Cage[+A]
```

- We use the – modifier to declare a type parameter as **contravariant**:

```
class Cage[-A]
```

Application of Variance Declarations

- We saw in the example of the Bird Cage and the Elephant that it is not always possible to declare a parameter as covariant or contravariant.
- Specifically, whether we can declare variance is dependent upon the *positions* where the type parameter is used.
  - Only in covariant positions (immutable) => Covariant
  - Only in contravariant positions (write-only) => Contravariant
  - Mixed (Read + Write) => Invariant

Co- and Contravariant Positions

Immutable Fields

```
class Cage[A] {
  val animal: A // covariant
}
```

Immutable fields (`val`) are in covariant position.

Co- and Contravariant Positions

Mutable Fields

```
class Cage[A] {
  var animal: A // covariant and contravariant
}
```

Mutable fields (`var`) are in both co- and contravariant position.

Co- and Contravariant Positions

Method Parameters

```
class Cage[A] {
  def put(animal: A): Unit // contravariant
}
```

Parameters to methods are in contravariant position.

Co- and Contravariant Positions

Method Return Types

```
class Cage[A] {
  def get: A // covariant
}
```

Method return types are in covariant position

- Note the common theme: reading imbues “covariance”, and writing imbues “contravariance”.
- Class parameters, unless promoted, don't have any variance applied to them.

Variance Declarations and the Compiler

This can seem complicated at first, but rest assured: Scala's compiler knows the rules and can help us along.

- **Example**: Let's try creating a Cage which accepts a covariant parameter, which is used in contravariant position, a method parameter:

```
class Cage[+A] {
  def put(a: A): Unit = ()
}
/*
 <console>:8: error: covariant type A occurs in
  contravariant position in type A of value a
 */
```

Exercise #11: Create a Queue & Use Covariance

Part I

- The standard library contains a *Queue* class, but we will write our own from scratch
- Create the *Queue* class in the *misc* package
  - Add an *A* type parameter and an *elements* class parameter of type immutable *Seq* of *A*
  - Override *equals* and *hashCode* delegating to *elements*
  - Override *toString* in case class fashion

Exercise #11: Create a Queue & Use Covariance

Part II

- Create the *Queue* companion object with an *apply* factory method
  - Which return type makes sense?
  - Add an *A* type parameter and *elements* repeated parameters of type *A*
  - Return a Queue containing all *elements*
- Make the constructor of *Queue* private

Exercise #11: Create a Queue & Use Covariance

Part III

- Add a *dequeue* method to *Queue*
  - Use a *Tuple2* of *A* and *Queue* of *A* for the return type
  - Return the first element and a new *Queue* without the dequeued element

  - Throw an *UnsupportedOperationException* if the *Queue* is empty

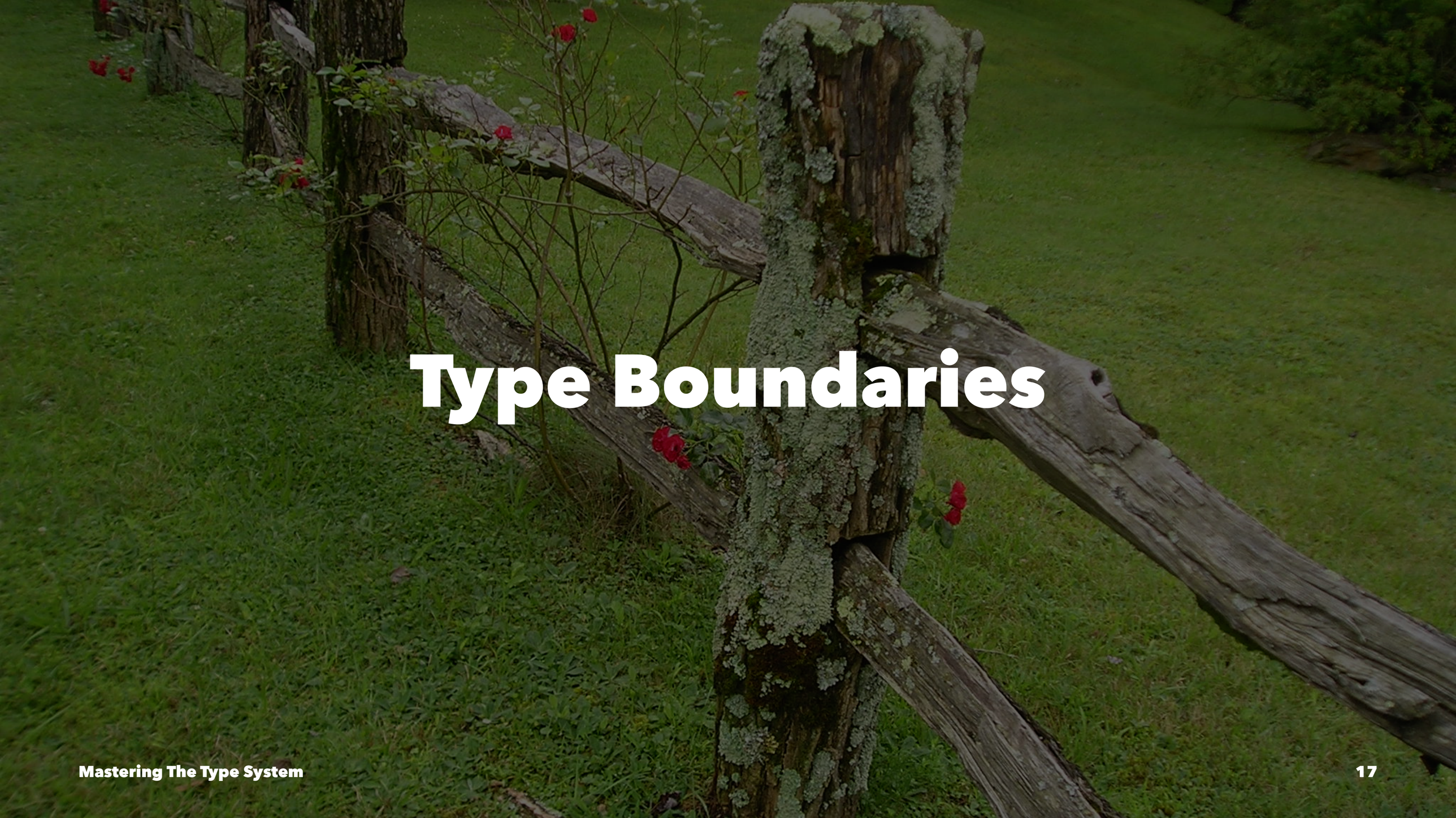
Covariance and Contravariant Positions

While covariance *seems* to make sense for our `Queue`, it would prevent us from adding an `enqueue` method:

```
class Queue[+A] {
  def enqueue(a: A): Queue[A]
  // ...
}
```

Adding an `enqueue` method would place ***A*** in contravariant position, and lead to the following compiler error: `covariant type A occurs in contravariant position in type A of value a`

- How could we work around this issue?



Type Boundaries

Lower Bounds

Declaring Lower Bounds

Declaring a Lower Bound expresses a relation of “must be a supertype of”. We declare a Lower Bound on a type parameter with the `>:` modifier:

```
case class Cage[A >: Animal](animal: A)
```

Lower Bounds

Usage

Scala will widen the type as much as necessary to meet the boundary:

```
Cage(new Bird)
// res0: Cage[Animal] = Cage(Bird@653e266e)

Cage("String")
// res1: Cage[Object] = Cage(String)

Cage(1)
// res2: Cage[Any] = Cage(1)
```

Exercise #12: Lower Bounds

- Add an *enqueue* method to *Queue*
  - Add a parameter of a suitable type (*A* does not work) for a new element to be enqueued
  - Return a new *Queue* with the new element enqueued at the end
  - Hint: Use a lower bounded type parameter at the method level

Upper Bounds

Declaring Upper Bounds

Declaring an *Upper* Bound expresses a relation of “is a”, and is declared with the `<:` modifier:

```
case class Cage[A <: Animal](animal: A)
```

Upper Bounds

Usage

The type argument is required to be a *subtype* of the Upper Bound:

```
Cage(new Bird)
// res0: Cage[Bird] = Cage(Bird@19b161e6)

Cage(new Animal)
// res1: Cage[Animal] = Cage(Animal@6bfc4d30)

Cage("Hello")
/* <console>:11: error: inferred type arguments [String]
     do not conform to method apply's
     type parameter bounds [A <: Animal] */
```

Digression: Package Objects

In Scala, packages are first-class citizens. Therefore, they can contain members. To define a package object, use the keywords `package object`:

```
package training.scala
package object air_scala {
  def isIncreasing...
  //...
}
```

- **Idiomatic Convention**: Package objects should be defined in `package.scala` files.
- Combined with chained package clauses, package objects can allow for very convenient scoping.

Exercise #13: Upper Bounds

- There is not necessarily a reason, why the `Itinerary.isScheduleIncreasing` method *has* to be restricted to `Itinerary/Seq[Flight]`.
- Create the package object `training.scala.air_scala` and add a version of the `isIncreasing` method
- Finally, make this new method work with *any* sequence of *Ordered* elements
  - **NOTE**: You may need to define your own instance of `Ordered[T]` to test with beside `Flight`.

A Final Word On Variance

Pandas and Heavy Metal


Type Members & Path Dependent Types

Types as Members

In addition to its usual “members” – fields and methods – classes, traits, singletons, and package objects can define **type members**.

To define a type member, we use the type keyword.

```
// an ‘abstract type member’
type Kilograms
// a ‘type alias’: aliasing a class to another name
type Kilograms = Double
// another ‘abstract type member’ but with an upper bound contract
type QualifiedPilot <: Pilot
```

Inner Types

Classes, traits singletons, and package objects can define inner types (which includes classes and traits):

```
class Outer {
  class InnerClass
  trait InnerTrait
  type InnerType = Kilograms
}
```

Path Dependent Types

Inner Types work somewhat differently in Scala than other languages. In particular, they are **path dependent**.

```
class Outer {
  class Inner
  def put(inner: Inner): Unit = ()
}
```

Path Dependent Types

We must have an outer instance (in this case, literally `Outer`) to reference an Inner Type:

```
val outer1 = new Outer
// outer1: Outer = Outer@65ba7645

val inner1: outer1.Inner = new outer1.Inner
// inner1: outer1.Inner = Outer$Inner@4435e0ec

val inner1: Outer.Inner = new outer1.Inner
// <console>:9: error: not found: value Outer
```

Path Dependent Types

The type of the inner instance is *dependent* upon the outer instance.

```
val outer2 = new Outer
// outer2: Outer = Outer@2049b67a

val inner2 = new outer2.Inner
// inner2: outer2.Inner = Outer$Inner@5ba5ad34

outer1.put(inner2)
/*
<console>:12: error: type mismatch;
 found   : outer2.Inner
 required: outer1.Inner
*/
```

This is the *path* to the type. The type of two inner instances from different outer instances - e.g. `inner1` and `outer2` - are not the same.

Type Selections

While they have different types, the inner instances of differing outer instances *do* have a common supertype:

```
val inner1: Outer#Inner = new outer1.Inner
// inner1: Outer#Inner = Outer$Inner@1c2ba649

val inner2: Outer#Inner = new outer2.Inner
// inner2: Outer#Inner = Outer$Inner@483242bb
```

This common supertype is denoted by a **type selection**, of `Outer#Inner`. This path is declared with the type selector, `#`.

Singleton Types

In Scala, every instance has a distinct type, known as the **singleton type**. We can use `.type` to reference a singleton type on an instance:

```
outer1.isInstanceOf[outer1.type]
// res0: Boolean = true

outer2.isInstanceOf[outer1.type]
// res1: Boolean = false
```

NOTE: `.type` can only be used where types are expected, e.g. `isInstanceOf`'s type argument.

Type Refinements

We can **refine** an existing type by declaring an extended type with additional declarations or definitions.

```
// Declare a new type: any `java.util.Date` subtype w/ `time: Long`
type EpochyDate = java.util.Date {
  def time: Long
}

// Anonymously instantiate a class, supplying the abstract member
new Runnable {
  override def run(): Unit =
    println("Running...")
}
```

A type declaration which adds a new member, rather than overriding an existing one, is known as a **structural type**.[§]

[§] Structural Types rely upon reflection; performance might suffer.

Static Duck Typing

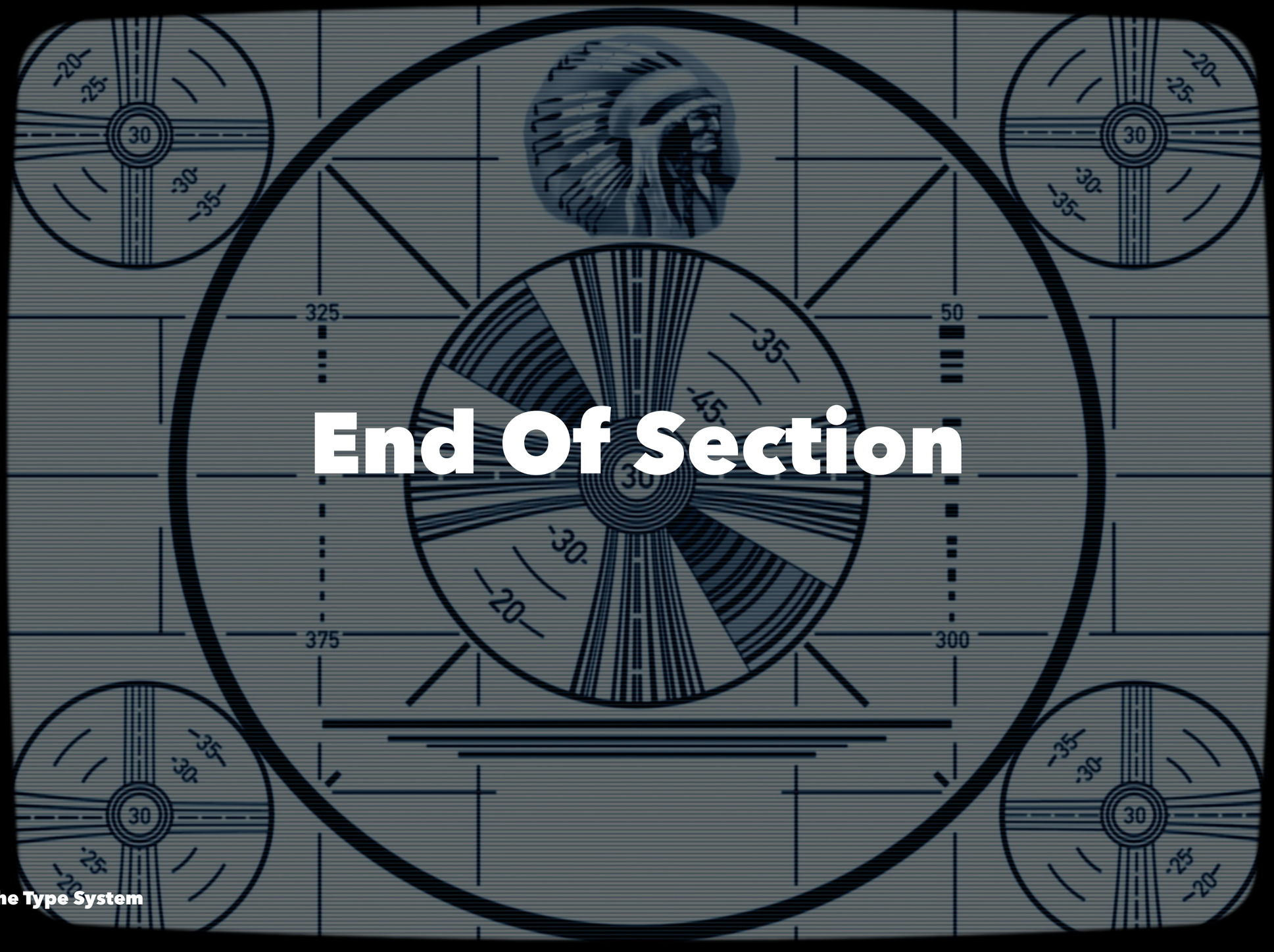
Omitting the type to be refined gives us a *purely* structural type, sometimes known as a **Duck Type**.

```
type Closeable = { def close(): Unit }

val c1: Closeable = new java.io.StringWriter
// c1: Closeable =

val c2: Closeable = new Door
// c2: Closeable = Door@4cda3058
```

The Duck Test: *"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."*


End Of Section